***Олимпиады школьников по информатике***

**Готовимся к олимпиаде по информатике**

***От редакции.***Задачи олимпиад по информатике, несмотря на большое разнообразие, очень часто можно отнести к той или иной теме. В данном выпуске мы рассмотрим три таких темы: динамическое программирование, алгоритмы на графах и вычислительную геометрию. Задачи на эти темы стали уже почти обязательным “олимпиадным минимумом”, хотя соответствующие алгоритмы не входят в школьную программу по информатике.

В настоящей тематической подборке использованы задачи различных соревнований. Материалы городской олимпиады (г. Нижний Новгород, 2004–2005 уч. г.) подготовлены предметной комиссией олимпиады под председательством В.Д. Лелюха. Задачи командных соревнований школьников Свердловской области по программированию и их решения были собраны в одноименной книге А.В. Ипатовым и любезно предоставлены редакции автором. Протестировать решение этих задач можно на замечательном сайте[*acm.timus.ru*](http://acm.timus.ru/). Кроме того, в подборке приведены задачи различных московских соревнований двух последних лет, тесты к которым можно найти на сайте [*www.olympiads.ru*](http://www.olympiads.ru/), а также задача Всероссийской олимпиады школьников по информатике 2005 г.

Все упомянутые задачи публикуются в нашей газете впервые. Мы надеемся, что данные материалы помогут педагогам в занятиях с одаренными школьниками, а будущим участникам соревнований — выступать на олимпиадах по информатике и программированию на равных с остальными.

***Динамическое программирование***

***Задача “Количество треугольников”***

**(Московская командная олимпиада 2005–2006 уч. г.)**

*Автор задачи* — *А.А. Петров
Автор разбора* — *В.М. Гуровиц*

Рассмотрим фигуру, аналогичную показанной на рисунке (большой равносторонний треугольник, составленный из маленьких равносторонних треугольников). На рисунке приведена фигура, состоящая из четырех уровней треугольников.

Напишите программу, которая будет определять, сколько всего в ней треугольников (необходимо учитывать не только “маленькие” треугольники, а вообще все треугольники — в частности, треугольник, выделенный жирным, а также вся фигура являются интересующими нас треугольниками).

 

**Формат входных данных**

Во входном файле записано одно число *N* — количество уровней в фигуре (1 *N*  100 000).

**Формат выходных данных**

Выведите в выходной файл количество треугольников в такой фигуре.

**Примеры**



**Решение**

Заметим, что все возможные треугольники — равносторонние, и одна из их сторон — горизонтальная. Такие треугольники могут быть двух типов: горизонтальная сторона снизу (треугольник ориентирован как вся фигура) или горизонтальная сторона сверху (треугольник ориентирован как выделенный треугольник на рисунке из условия).

Подсчитаем сначала количество треугольников первого типа. Заметим, что каждый такой треугольник однозначно задается своим основанием, а значит, парой его концов — узлов сетки, лежащих на одной горизонтали. Если в некоторой горизонтали *k* узлов, то пару точек можно выбрать *k*(*k* – 1)/2 способами, то есть существуют *k*(*k* – 1)/2 треугольников с таким основанием. Если фигура состоит из *n* уровней, то в верхней горизонтали имеются 2 узла, в следующей — 3, …, в последней — *n*(*n* + 1)/2. Таким образом, количество треугольников первого типа равно 1/2(1·2 + 2·3 + ... +n(n+1). Для предложенных в задаче ограничений на входные данные можно вычислить эту сумму, используя цикл. Но можно вычислить эту сумму в явном виде, не используя компьютер.

Чуть сложнее подсчитать количество треугольников второго типа. Покажем один из возможных способов это сделать. Для удобства обозначим *Sn* = 1 + 2 + 3 + … + *n* = *= n*(*n* + 1)/2. Подсчитаем отдельно количество треугольников со стороной 1, со стороной 2, со стороной 3 и т.д. Из рисунка видно, что эти количества равны *Sn-1*, *Sn-3*, *Sn-5*, … соответственно (слагаемые каждой из сумм*Si* — это количества треугольников соответствующего размера в одном горизонтальном ряду). Таким образом, общее количество треугольников второго типа равно *Sn-1* + *Sn-3* + *Sn-5* + … = (*n*(*n* – 1) + (*n* – 2)(*n* – 3) + (*n* – 4)(*n* – 5)+ + …)/2 (последнее слагаемое в этой сумме равно *S*2 или *S*1 в зависимости от четности *n*). Эту сумму также можно сосчитать, используя цикл, а можно упростить, записав ответ в явном виде. Общее количество треугольников обоих типов равно *k*(*k* + 1)(4*k* + 1)/2, если *n* = 2*k*, и (*k* + 1)(4*k*2 + 7*k* + 2)/2, если  *n* = 2*k* + 1. Доказать это можно по индукции.

***Задача “Представление числа”***

**(Московская командная олимпиада 2005–2006 уч. г.)**

 *Автор задачи и разбора* — *А.Ю. Гусаков*

Учительница математики попросила школьников составить арифметическое выражение так, чтобы его значение было равно данному числу *N*, и записать его в тетради. В выражении могут быть использованы натуральные числа, не превосходящие *K*, операции сложения и умножения, а также скобки. Петя очень не любит писать и хочет придумать выражение, содержащее как можно меньше символов. Напишите программу, которая поможет ему в этом.

**Формат входных данных**

В первой строке входного файла записаны два натуральных числа: *N* (1  *N* 10 000) — значение выражения и *K* (1  *K*  10 000) — наибольшее число, которое разрешается использовать в выражении.

**Формат выходных данных**

В единственной строке выходного файла выведите выражение с данным значением, записывающееся наименьшим возможным количеством символов.

Если решений несколько, выведите любое из них.

**Примечание**

При подсчете длины выражения учитываются все символы: цифры, знаки операций, скобки.

**Примеры**



**Решение**

Прежде чем перейти к описанию идеи решения, введем несколько определений.

Будем называть выражение *C* *выражением первого типа*, если оно является числом от 1 до *K*, или если последней операцией при его вычислении является сложение, то есть *C* = *A* + *B*, где *A* и *B* — некоторые выражения.

Будем называть выражение *C* *выражением второго типа*, если оно является числом от одного до *K* или если последней операцией при его подсчете является умножение, то есть  *C* = *A*·*B*, где *A* и *B* — некоторые выражения. Например,
(2 + 3·4) + 5·7 — выражение первого типа, (2 + 3)·(3 + 2) — выражение второго типа, 1 — выражение, относящееся как к первому, так и ко второму типу.

Теперь обсудим идею решения. Пусть a[1,m] равно наименьшей длине выражения первого типа, значение которого равно m; a[2,m] равно наименьшей длине выражения второго типа, значение которого равно m. Легко видеть, что при m  K значения a[1,m] и a[2,m] равны длине числа m. Пусть теперь m > K, и у нас уже вычислены все значения a[1,1], …, a[1,m-1], a[2,1], …,
a[2,m-1].

Сначала вычислим a[1,m]. Поскольку соответствующее выражение представимо в виде *C* = *A* + *B*, нам достаточно перебрать все возможные значения и типы выражений *A* и *B* и выбрать оптимальную комбинацию. Иными словами, a[1,m] = min(a[i1,t] + a[i2,m - t] + 1), где i1, i2 — 1 или 2, а t  m/2 (можно считать, что значение *A* меньше или равно *m*/2, иначе *A* и *B* можно поменять местами). В переменных how[1,m][1], how[1,m][2], how[1,m][3] будем хранить те значения i1, t, i2 соответственно, при которых достигается минимальное значение.

Теперь вычислим a[2,m]. Соответствующее выражение представимо в виде *A*·*B*, где *A* и *B* — либо выражения второго типа, либо выражения первого типа, взятые в скобки (либо какая-то их комбинация). Будем считать, что значение *A* меньше или равно, иначе поменяем местами *A* и *B*. Таким образом, a[2,m] = min(a[i1,t] +
+ a[i2,m/t] + 1 + 2\*(4-i1-i2)), где i1, i2 — 1 или 2, а t  и m делится на t.

*Замечание*. Число 4 - i1 - i2 равно количеству выражений первого типа среди *A* и*B*, а значит, равно количеству пар скобок, которое придется добавить.

В переменных how[2,m][1], how[2,m][2], how[2,m][3] будем хранить те значения i1, t, i2 соответственно, при которых достигается минимум.

Теперь мы умеем вычислять значения a[1,1], ..., a[1,N], a[2,1], ..., a[2,N]. Естественно, что искомое выражение является выражением первого или второго типов, поэтому его длина равна min(a[1,N], a[2,N]). Осталось только научиться его выводить. Для этого напишем рекурсивную процедуру Save(last, n), которая будет выписывать выражение типа last, значением которого является n. Во-первых, если n  K, то нужно просто вывести число cur. Иначе нужно посмотреть на значение last.

Если last = 1 (то есть выражение имеет вид *A* + *B*), то сначала выпишем первое слагаемое — Save(how[last,n][1], how[last,n][2]), затем поставим знак ‘+’ и выпишем второе слагаемое — Save(how[last,n][3], n - how[last,n][2]).

Если last = 2 (то есть имеем дело с произведением), то нужно действовать аналогично случаю last = 1, но при этом не забыть, что если какой-то из множителей первого типа, то его надо окружить скобками.

Количество операций, выполняемых алгоритмом, пропорционально *N*2.

Оказывается, что приведенную динамическую схему можно упростить. Достаточно для каждого числа m хранить информацию лишь о самом коротком выражении, независимо от его типа. При этом, если для данного числа самое короткое выражение может быть как выражением типа 1, так и выражением типа 2, то нужно выбирать произведение, а не сумму. Обоснование правильности этого алгоритма оставим читателю.

***Задача “Кафе”***

**(Московская командная олимпиада 2004–2005 уч. г.)**

*Автор задачи — А.П. Лахно*
*Автор разбора*—*В.Ю. Антонов*

Около Петиного университета недавно открылось новое кафе, в котором действует следующая система скидок: при каждой покупке более чем на 100 рублей покупатель получает купон, дающий право на один бесплатный обед (при покупке на сумму 100 рублей и меньше такой купон покупатель не получает).

Однажды Пете на глаза попался прейскурант на ближайшие *N* дней. Внимательно его изучив, он решил, что будет обедать в этом кафе все *N* дней, причем каждый день он будет покупать в кафе ровно один обед. Однако стипендия у Пети небольшая, и поэтому он хочет по максимуму использовать предоставляемую систему скидок так, чтобы его суммарные затраты были минимальны. Требуется найти минимально возможную суммарную стоимость обедов и номера дней, в которые Пете следует воспользоваться купонами.

**Формат входных данных**

В первой строке входного файла записано целое число *N* (0  *N*  100). В каждой из последующих *N* строк записано одно целое число, обозначающее стоимость обеда в рублях на соответствующий день. Стоимость — неотрицательное целое число, не превосходящее 300.

**Формат выходных данных**

В первой строке выдайте минимальную возможную суммарную стоимость обедов. Во второй строке выдайте два числа: *K*1 и *K*2 — количество купонов, которые останутся неиспользованными у Пети после этих *N* дней, и количество использованных им купонов соответственно.

В последующих *K*2 строках выдайте в возрастающем порядке номера дней, когда Пете следует воспользоваться купонами. Если существует несколько решений с минимальной суммарной стоимостью, то выдайте то из них, в котором значение *K*1 максимально (на случай, если Петя когда-нибудь еще решит заглянуть в это кафе). Если таких решений несколько, выведите любое из них.

**Пример**



**Решение**

Заведем двумерный массив *A*, в ячейке *A*[*i*,*j*] которого будет храниться минимальная сумма, которую мог потратить Петя за первые *i* дней, при условии, что после этого у него в запасе осталось *j* талонов на бесплатную еду. Положим*A*[0,0] = 0; *A*[0,*i*] =  при *i* > 0. Пусть *P*[*i*] — цена обеда в *i*-й день. Для заполнения ячейки *A*[*i*,*j*] можно воспользоваться следующей формулой:



Ответом будет являться минимальное из чисел (назовем его *Ans*), хранящихся в*N*-м столбце. Рассмотрим все элементы *N*-го столбца, значения которых совпадают с *Ans*. Понятно, что *K*1 — максимальный номер строки из всех номеров строк этих элементов. Как же посчитать параметр *K*2, который также нужно вывести? Для этого дополнительно будем хранить в *B*[*i*,*j*] общее количество купонов на бесплатные обеды, полученных Петей (считается по формулам, сходным с формулами для вычисления *A*[*i*,*j*]). Можно не заводить второй массив,
а считать *K*2 непосредственно перед выводом ответа.

Остается вывести дни, в которые были потрачены купоны. Для ячейки массива*A*[*i*,*j*] по числам в столбце *A*[*i*–1] можно определить, был ли использован купон в*i*-й день. А зная *K*1, можно вывести все дни, в которые были потрачены купоны, и подсчитать их количество *K*2.

***Задача “Скобки”***

**(Московская командная олимпиада 2004–2005 уч. г.)**

*Авторы задачи — Р.А. Жуйков, М.О. Трухина
Автор разбора*—*В.Ю. Антонов*

Назовем строку *S* правильной скобочной последовательностью, если она состоит только из символов ‘{’, ‘}’, ‘[’, ‘]’, ‘(’, ‘)’ и выполнено хотя бы одно из следующих трех условий:

1) *S* — пустая строка;

2) *S* можно представить в виде *S* = *S*1 + *S*2 + *S*3 + ...+ + *SN* (*N* > 1), где *Si* — непустые правильные скобочные последовательности, а знак ‘+’ обозначает конкатенацию (приписывание) строк;

3) *S* можно представить в виде *S* = ‘{’ + *C* + ‘}’ или *S* = ‘[’ + *C* + ‘]’ или *S* = ‘(’ + *C* + ‘)’, где *C* является правильной скобочной последовательностью.

Дана строка, состоящая только из символов ‘{’, ‘}’, ‘[’, ‘]’, ‘(’, ‘)’. Требуется определить, какое минимальное количество символов надо вставить в эту строку для того, чтобы она стала правильной скобочной последовательностью.

**Формат входных данных**

В первой строке входного файла записана строка, состоящая только из символов ‘{’, ‘}’, ‘[’, ‘]’, ‘(’, ‘)’. Длина строки не превосходит 100 символов.

**Формат выходных данных**

Вывести в первую строку выходного файла единственное неотрицательное целое число — ответ на поставленную задачу.

**Примеры**

|  |  |
| --- | --- |
| **Входной файл** | **Выходной файл** |
| {(}) | 2 |
| ([{}]) | 0 |

**Решение**

Пусть задана строка *S*, состоящая из символов ‘{’, ‘}’, ‘[’, ‘]’, ‘(’, ‘)’. Заведем двумерный массив *A*, в элементе *A*[*i*,*j*] которого будем хранить минимальное количество скобок, которые нужно добавить к подстроке строки *S* с *i-*го по *j*-й символ (*S*[*i*]..*S*[*j*]), чтобы получить правильную скобочную последовательность. Если *i* > *j*, то положим *A*[*i*,*j*] = 0. Будем находить решения, постепенно увеличивая разность между *j* и *i*, учитывая, что *A*[*i*,*i*] = 1 для любого *i*(подумайте, почему). При увеличении разности между *j* и *i* вычисление *A*[*i*,*j*] осуществляется по следующим правилам:

1) если *S*[*i*] — закрывающая скобка, то в правильной скобочной последовательности до нее должна быть открывающая скобка, т.е. *A*[*i*,*j*] = 1 +*A*[*i*+1,*j*];

2) если *S*[*i*] — открывающая скобка, то возможно несколько вариантов восстановления правильной скобочной последовательности:

а) среди остальных скобок существует пара для этой, тогда:

*A*[*i*,*j*] = *min*(*A*[*i*+1,*k*–1] + *A*[*k*+1,*j*]), где минимум берется по всем *k*, таким, что *S*[*k*] — закрывающая скобка, соответствующая открывающей скобке *S*[*i*];

б) соответствующую закрывающую скобку нужно добавить к строке, т.е. *A*[*i*,*j*] = *min*(*A*[*i*+1,*k*]+ *A*[*k*+1,*j*]) при *i* Ј *k* Ј *j* (нужную закрывающую скобку вставляем после *k*-й скобки).

*A*[*i*,*j*] равен минимуму из этих двух вариантов.

Ответом будет являться *A*[1,*Length*(*S*)].

***Задача “Числа”***

**(I городская олимпиада, г. Нижний Новгород, 2004–2005 уч. г.)**

Дана последовательность чисел *a*1, *a*2, …, *aN*. За одну операцию разрешается удалить любое (кроме крайних) число, заплатив за это штраф, равный произведению этого числа на сумму соседних. Требуется удалить все числа, кроме крайних, с минимальным суммарным штрафом.

Например.

Начальная последовательность: 1 50 51 50 1

Удаляем четвертое число, штраф 50(1 + 51) = 2600, получаем 1 50 51 1

Удаляем третье число, штраф 51(50 + 1) = 2601, получаем 1 50 1

Удаляем второе число, штраф 50(1 + 1) = 100.

Итого штраф 5301.

**Формат входных данных**

В первой строке входного файла расположено одно число *N* (1  *N*  100) — количество чисел в последовательности.

Во второй строке находятся *N* целых чисел *a*1, *a*2, …, *a*N; никакое из чисел не превосходит по модулю 100.

**Формат выходных данных**

Выведите в выходной файл одно число — минимальный суммарный штраф.

**Пример**

|  |  |
| --- | --- |
| **Входной файл** | **Выходной файл** |
| 5 1 50 51 50 1 | 5301 |

**Решение**

Заведем квадратную матрицу *M* размера *N* x *N*. В ячейке *Mij* будем хранить минимальный штраф за удаление всех чисел с *i*-го по *j*-е не включительно (элементы матрицы, у которых *i* і *j*, нам не будут нужны). Подсчитывать эти штрафы будем следующим образом: поскольку среди чисел между *i*-м и *j*-м какое-то (*k*-е) мы должны удалить последним, то переберем все *k* от *i* до *j* не включительно и посчитаем наименьший штраф за удаление всех чисел между *i* и*j* при условии, что *k*-е удаляется последним. Этот штраф равен сумме наименьших штрафов за удаление всех чисел *c* *i* по *k* (*Mik*), с *k* по *j* (*Mkj*) и штрафа за последнее удаление — (*ai* + *aj*)*ak*. Тогда очевидно, что



Несложно посчитать элементы матрицы с индексами, отличающимися на 1: так как удалять между соседними числами нечего, то *Mi,i+1* = 0.

Осталось только вычислить все элементы матрицы (например, в порядке увеличения разности между индексами) и посмотреть на число *M1N*, которое и будет ответом на задачу.

Кроме того, надо аккуратно обработать случай *N* = 1. Вышеприведенный алгоритм для этого случая не подходит, но очевидно, что, так как удалять при *N*= 1 нечего, ответ здесь — 0.

***Задача “Казино”***

**(XVIII Всероссийская олимпиада школьников по информатике, 2005 г.)**

Вновь открытое казино предложило оригинальную игру.

В начале игры крупье выставляет в ряд несколько фишек разных цветов. Кроме того, он объявляет, какие последовательности фишек игрок может забирать себе в процессе игры. Далее игрок забирает себе одну из заранее объявленных последовательностей фишек, расположенных подряд. После этого крупье сдвигает оставшиеся фишки, убирая разрыв. Затем игрок снова забирает себе одну из объявленных последовательностей и так далее. Игра продолжается до тех пор, пока игрок может забирать фишки.

Рассмотрим пример. Пусть на столе выставлен ряд фишек *rrrgggbbb*, и крупье объявил последовательности *rg* и *gb*. Игрок, например, может забрать фишки *rg*, лежащие на третьем и четвертом местах слева. После этого крупье сдвинет фишки, и на столе получится ряд *rrggbbb*. Еще дважды забрав фишки *rg*, игрок добьется того, что на столе останутся фишки *bbb* и игра закончится, так как игроку больше нечего забрать со стола. Игрок мог бы действовать и по-другому — на втором и третьем ходах забрать не последовательности *rg*, а последовательности *gb*. Тогда на столе остались бы фишки *rrb*. Аналогично, игрок мог бы добиться того, чтобы в конце остались ряды *rrr* или *rbb*.

После окончания игры полученные фишки игрок меняет на деньги. Цена фишки зависит от ее цвета.

Требуется написать программу, определяющую максимальную сумму, которую сможет получить игрок.

**Формат входных данных**

В первой строке входного файла записано число *K* (1  *K*  26) — количество цветов фишек. Каждая из следующих *K* строк начинается со строчной латинской буквы, обозначающей цвет. Далее в той же строке через пробел следует целое число *Xi* (1  *Xi*  150, *i* = 1..*K*) — цена фишки соответствующего цвета.

В (*K* + 2)-й строке описан ряд фишек, лежащих на столе в начале игры. Ряд задается *L* строчными латинскими буквами (1  *L*  150), которые обозначают цвета фишек ряда.

В следующей строке содержится число *N* (1  *N* 150) — количество последовательностей, которые были объявлены крупье. В следующих *N* строках записаны эти последовательности. Гарантируется, что сумма длин этих *N* строк не превосходит 150 символов и все они непустые.

**Формат выходных данных**

В выходной файл выведите единственное целое число — максимальную сумму денег, которую может получить игрок.

**Пример**



**Решение**

Во-первых, заметим, что не любая стратегия игрока приводит к нужному результату. Например, если в примере, разбиравшемся в условии задачи (*rrrgggbbb*), стоимость буквы *b* будет больше стоимости буквы *r*, то только один из путей приводит к оптимальному результату: три раза забрать буквы *gb*. После некоторого размышления можно понять, что всевозможные так называемые “жадные” решения — делать самый выгодный ход, делать самый длинный ход, делать самый левый ход и т.п. — также в большинстве случаев неверны. Перебирать же все варианты, для каждого из них — опять все варианты, и так далее — оказывается слишком медленным подходом.

Для решения этой задачи применим динамическое программирование. В первой половине решения мы для каждого куска начальной последовательности фишек определим, можем ли мы забрать его целиком, не затрагивая при этом остальные фишки. Иными словами, для всех *l* и *r* найдем величину *a*[*l*,*r*], равную 1, если фишки с номерами (то есть позициями в начальном расположении, считая слева) с *l* по *r* игрок может за несколько действий все забрать себе, и 0 — в противном случае. Затем по этим данным во второй половине решения мы восстановим ответ к задаче.

Во избежание путаницы будем называть последовательности, объявленные крупье (в отличие от начальной последовательности), *словами*.

Вторая половина решения проще первой, поэтому начнем с нее. Пусть нам известны величины *a*[*l*,*r*]. Пусть тогда *b*[*l*,*r*] — это максимальная сумма денег, которую можно получить, играя только на отрезке с *l*-й фишки по *r*-ю (то есть не трогая остальные фишки). Ясно, что если *a*[*l*,*r*] = 1, то *b*[*l*,*r*] — это просто сумма стоимостей всех фишек с *l*-й по *r*-ю. Если же *a*[*l*,*r*] = 0, то уже не удастся забрать все фишки, соответственно, какая-то из фишек должна остаться — пусть это фишка с номером *k*. Тогда заметим, что задача распадается на две — для фишек слева от *k*-й и для фишек справа от нее, т.е. что *b*[*l*,*r*] = *b*[*l*,*k*– 1] + *b*[*k*+ 1,*r*] (здесь и далее мы считаем, что для любого *mb*[*m*,*m*– 1] = 0). Но так как номер оставшейся фишки не фиксирован, то в действительности *b*[*l*,*r*] равно максимуму этих величин по всем *k*:

(\*)

(напомним, что эта формула верна при *a*[*l*,*r*] = 0). Теперь мы видим, что для нахождения величин *b*[*l*,*r*] можно применить динамическое программирование: если мы будем перебирать *r* от 1 до длины начальной последовательности, а затем *l* от *r* до 1, то к моменту вычисления величины *b*[*l*,*r*] все величины, которые необходимы для ее нахождения по формуле (\*), уже вычислены. Таким образом, зная величины *a*[*l*,*r*], мы можем найти *b*[*l*,*r*]. Легко видеть, что на этом шаге требуется порядка *L*3 действий (напомним, что *L* — это длина начальной последовательности).

Теперь вернемся к первой половине решения, а именно — нахождению величин*a*[*l*,*r*]. Как узнать, можно ли забрать себе данную последовательность (в нашем случае — кусок начальной) целиком? Во-первых, возможно, эту последовательность можно разделить на две части, каждую из которых можно забрать себе целиком. То есть, если найдется *k* такое, что *a*[*l*,*k*] = 1 и *a*[*k*+ 1,*r*] = 1, то и *a*[*l*,*r*] = 1. Если же так сделать не удается, то можно заметить, что если мы все-таки заберем всю последовательность целиком, то ее первую и последнюю фишки мы заберем только на последнем шаге. Пусть на последнем шаге мы заберем некоторое слово *S*. Тогда задача сводится к тому, чтобы выкинуть из последовательности некоторые куски (можно считать, что для этих кусков величина *a* уже посчитана, поэтому мы знаем, какие из них можно выкинуть, а какие — нет), оставив первую и последнюю фишки на месте, так, чтобы осталось слово *S*.

Для решения этой задачи опять применим динамическое программирование. А именно, пусть *c*[*l*,*r*,*p*,*q*] = 1, если из отрезка с *l*-й фишки по *r*-ю можно так выкинуть несколько кусков, чтобы остались первые *q* фишек слова номер *p*, и при этом *l*-я и *r*-я фишки остались на месте, и 0 — иначе (тогда нас интересует величина *c*[*l*,*r*,*p*,*length*(*p*)], где *length*(*p*) — длина слова номер *p*). Во-первых, ясно, что если первая фишка слова номер *p* не совпадает с *l*-й фишкой начальной последовательности, или *q*-я — с *r*-й, то *c*[*l*,*r*,*p*,*q*] = 0. В противном случае пусть (*q*– 1)-я фишка слова получится из  *k*-й фишки последовательности. Тогда, во-первых, должно быть *c*[*l*,*k*,*p*,*q*– 1] = 1 (чтобы получить первые  *q* – 1 фишек), а, во-вторых, *a*[*k*+ 1,*r*– 1] = 1 (чтобы выкинуть кусок между (*q* – 1)-й фишкой и *q*-й). Но так как число *k* не фиксировано, то получаем, что*c*[*l*,*r*,*p*,*q*] = 1 тогда и только тогда, когда найдется такое *k*. Тем самым получен алгоритм для подсчета *c*. Оценим время его работы. На подсчет одного элемента массива *c* требуется порядка *L* действий (по количеству возможных *k*). Нам необходимо подсчитать элементы вида *c*[*l*,*r*,*p*,*length*(*p*)]. Из вышеприведенных рассуждений легко заметить, что для их подсчета нам потребуется подсчет только элементов вида *c*[*l*,…,…,…], а элементов такого вида порядка *L*Ч*S* (где *S*— сумма длин всех слов), так как для параметра *r* есть порядка *L* вариантов, а для пары параметров (*p*, *q*) — порядка *S* вариантов. Тем самым мы для любых *l* и*r* можем найти *a*[*l*,*r*] за порядка *L*·*L*·*S* = *L*2·*S* действий, а общее время работы получается порядка *L*2·*L*2·*S* = *L*4·*S*.

Однако легко заметить, что если подсчитать все величины *c*[*l*,*r*,*p*,*q*] сначала, а потом лишь использовать найденные значения, то время работы будет меньше. Действительно, всего в массиве *c* *L*2·*S* элементов, и на подсчет каждого из них уйдет порядка *L* действий, тем самым общее время работы будет порядка *L*3·*S*.

***Алгоритмы на графах***

***Задача “День объединения”***

**(I командное соревнование школьников Свердловской области по программированию, 2000 г.)**

*Автор задачи — А.Ботов
Автор разбора — А.Ипатов*

Исландия — страна, расположенная на множестве мелких островков. Прошло уже много времени с того момента, когда был построен первый мост. Сейчас это государство полностью соединено мостами, и с любого острова можно попасть на любой другой, не пользуясь лодкой. Недавно жители решили добавить к уже большому списку праздников еще один — День Объединения. Как вы, наверное, уже догадываетесь, так они решили назвать тот день, в который жители перестали нуждаться в лодках для перемещения между любыми двумя островами. К сожалению, было построено очень много мостов, и разобраться, когда же настал этот день, оказалось не совсем просто.

Жители Исландии будут вам очень благодарны, если вы напишете программу, которая определит по датам постройки мостов, когда же наступил этот знаменательный день. Итак, надо определить тот день, в который все острова оказались соединенными (пусть даже через другие острова) мостами.

**Формат входных данных**

Во входном файле для этой задачи вы сможете найти: в первой строке два числа *N* и *K* — количество островов и мостов соответственно (2  *N*  1000, 1  *K*  10 000). В последующих *K* строках вам будет дана информация о мостах в следующем виде: “*a* *b* *date*”, где *date* — дата в формате *dd*/*mm*/*yyyy*, а числа *a* и *b*— номера островов, которые соединил построенный мост. Записи в файле упорядочены по времени, последний построенный мост — в конце файла. Гарантируется, что после постройки последнего моста в Исландии с любого острова можно попасть на любой другой, пользуясь только мостами (возможно несколькими).

**Формат выходных данных**

Выходной файл должен содержать единственную дату, записанную в том же формате, что и во входном файле, соответствующую тому дню, когда жители Исландии смогли ходить друг к другу в гости пешком.

**Пример**



**Решение**

Рассмотрим все острова и мосты между ними (такая модель в математике называется графом, острова называются вершинами графа, а мосты — ребрами графа). Будем говорить, что группа островов связна, если с любого острова группы можно попасть на любой другой ее остров, пользуясь только мостами. Так, все острова можно разбить на связные группы (назовем их компонентами связности), причем до построения первого моста каждый остров представлял собой отдельную компоненту связности, а после построения последнего все острова лежат в одной компоненте связности. Назовем остров изолированным, если с него нельзя попасть ни на какой другой остров, пользуясь только мостами. В начале все острова являются изолированными. Назовем вновь построенный мост *ab* полезным, если до его постройки нельзя было попасть с острова *a* на остров *b*, пользуясь только мостами. В противном случае назовем мост *ab* бесполезным. Построение каждого полезного моста уменьшает количество компонент связности ровно на 1, следовательно, всего полезных мостов *N* – 1, и День Объединения наступает после построения последнего полезного моста. Рассмотрим алгоритм раскраски, решающий данную задачу: заведем массив из *N* элементов *color*[*i*], указывающий цвет вершины(острова) *i*— номер компоненты связности, в которой она лежит. При этом будем нумеровать только компоненты, состоящие более чем из одной вершины, то есть для изолированных вершин в массиве *color* будет стоять 0. Затем последовательно считываем мосты, определяя, является ли следующий мост полезным, или нет. Мост *ij* является бесполезным, если *color*[*i*] = *color*[*i*] > 0. Иначе увеличиваем счетчик полезных мостов *useful*. При этом выполняем следующие действия: если обе вершины не покрашены, то красим их цветом с номером *useful*. Если одна вершина покрашена, а вторая — нет, то красим вторую цветом первой. Если обе вершины покрашены, но в разные цвета, красим все вершины, покрашенные в цвет второй, цветом первой.

Фактически мы воспользовались алгоритмом построения остова в графе.

***Задача “Генеалогическое дерево”***

**(II командное соревнование школьников Свердловской области по программированию, 2000 г.)**

*Автор задачи — Л.Волков
Автор разбора — А.Ипатов*

Система родственных отношений у марсиан достаточно запутана. Собственно говоря, марсиане почкуются когда им угодно и как им угодно, собираясь для этого разными группами, так что у марсианина может быть и один родитель, и несколько десятков, а сотней детей сложно кого-нибудь удивить. Марсиане привыкли к этому, и такой жизненный уклад кажется им естественным.

А вот в Планетарном Совете запутанная генеалогическая система создает серьезные неудобства. Там заседают достойнейшие из марсиан, и поэтому, чтобы никого не обидеть, во всех обсуждениях слово принято предоставлять по очереди, так, чтобы сначала высказывались представители старших поколений, потом те, что помладше, и лишь затем уже самые юные и бездетные марсиане. Однако соблюдение такого порядка на деле представляет собой совсем не простую задачу. Не всегда марсианин знает всех своих родителей, что уж тут говорить про бабушек и дедушек! Но когда по ошибке сначала высказывается праправнук, а потом только молодо выглядящий прапрадед — это настоящий скандал.

Ваша цель — написать программу, которая определила бы раз и навсегда такой порядок выступлений в Планетарном Совете, который гарантировал бы, что каждый член Cовета получает возможность высказаться раньше любого из своих потомков.

**Формат входных данных**

В первой строке входного файла к этой задаче находится единственное число*N*, 1  *N*  100*—* количество членов Марсианского Планетарного Совета. По многовековой традиции все члены Совета нумеруются натуральными числами от 1 до *N*. Далее во входном файле следуют ровно *N* строк, причем *I*-я строка содержит список детей члена Совета с порядковым номером *I*. Список детей представляет собой последовательность порядковых номеров детей, разделенных пробелами и следующих в произвольном порядке. Список детей может быть пустым. Список детей (даже если он пуст) оканчивается нулем.

**Формат выходных данных**

Выходной файл должен в своей единственной строке содержать последовательность номеров выступающих, разделенных пробелами. Если несколько последовательностей удовлетворяют условиям задачи, то можно вывести любую из них.

**Пример**



**Решение**

Проще найти обратный порядок выступления, то есть такой порядок, что любой марсианин выступает в Совете позже своих потомков. Поймем, что в любой группе марсиан есть марсианин, не имеющий детей. Если бы это было не так, мы бы взяли цепочку марсиан *A*1, *A*2, *A*3, …, *A*k, …, где *A*k — сын *A*k-1. В силу того, что марсиан конечное число, эта цепочка оборвется на *A*m. Марсианин *A*mне имеет детей.

Теперь возьмем марсианина, не имеющего детей, и заставим его выступать первым. В оставшейся группе вновь найдем марсианина, не имеющего детей среди марсиан этой группы, и заставим его выступать вторым и т.д. Всегда будет получаться, что имеющий детей марсианин выступает в Совете после них. Данный алгоритм решает задачу топологической сортировки вершин в ориентированном графе родственных отношений.

После того, как мы найдем такой порядок выступления марсиан, обратим его и получим нужный нам порядок.

Есть несколько реализаций этого алгоритма, использующих различные структуры данных. Мы для работы будем использовать такую структуру, как матрица смежности графа, — двумерный массив логического



Эта реализация не является самой быстрой, но зато очень проста в понимании. Заведем еще массив *used*[*i*], отмечающий, выступал ли уже марсианин с номером *i*, или нет, и массив *n*\_*of*\_*sons*[*i*], показывающий количество детей у *i*-го марсианина. После ввода данных матрица смежности заполнена, и *n*\_*of*\_*sons*[*i*] =*количеству элементов true в строке* *son*[*i*]. Приведем основной фрагмент решения задачи:

**for** i := 1 **to** n **do**

**begin**

**for** j := 1 **to** n **do**

**if** **not** used[j] **and** (n\_of\_sons[j] = 0)

**then** **break;**

      backorder[i] := j;

      used[j] := true;

**for** k := 1 **to** n **do**

**if** son[k][j] **then**

**begin**

               dec(n\_of\_sons[k]);

               son[k][j] := false

**end**;

**end**;

**for** i := n **downto** 1 **do**

write(backorder[i],' ');

***Задача “Метро”***

**(IV командное соревнование школьников Свердловской области по программированию, 2001 г.)**

*Автор задачи — Л.Волков
Автор разбора — А.Ипатов*



Многие программисты “СКБ Контур” любят добираться до работы на метро — благо головной офис расположен совсем недалеко от станции “Уралмаш”. Ну а поскольку сидячий образ жизни требует активных физических нагрузок в свободное от работы время, многие сотрудники — в том числе и Никифор — ходят от дома до метро пешком.

Никифор живет в таком районе нашего города, где улицы образуют правильную сетку кварталов; все кварталы являются квадратами с длиной стороны, равной 100 метрам. Вход на станцию метро расположен на одном из перекрестков; Никифор начинает свой путь с другого перекрестка, который расположен южнее и западнее входа в метро. Естественно, что, выйдя из дома, Никифор всегда идет по улицам, ведущим либо на север, либо на восток. Некоторые кварталы, которые встречаются ему на пути, он может также пересечь по диагонали, ведущей из юго-западного угла квартала в северо-восточный. Таким образом, некоторые из маршрутов (ведущих всегда на север, восток или северо-восток) оказываются короче других. Никифора интересует, сколько времени понадобится ему на преодоление кратчайшего маршрута; для этого ему нужно знать его длину. Вы должны написать программу, которая по имеющейся информации о виде сетки кварталов рассчитывает длину кратчайшего маршрута из юго-западного угла в северо-восточный.

**Формат входных данных**

В первой строке входного файла находятся два целых числа *N* и *M* (0 < *N*, *M* 1000) — размер сетки кварталов с запада на восток и с юга на север соответственно. Никифор начинает путь с перекрестка, который находится к юго-западу от квартала с координатами (1, 1); станция метро находится к северо-востоку от квартала с координатами (*N*, *M*). Во второй строке входного файла находится целое число *K* (0  *K*  100) — количество кварталов, через которые можно пройти наискось. Далее следуют *K* строк с парами натуральных чисел, разделенных пробелами — координатами таких кварталов.

**Формат выходных данных**

В выходной файл требуется вывести длину кратчайшего пути от дома Никифора до станции метро в метрах, округленную до целых метров.

**Пример**



**Решение**

Переведем эту задачу на язык графов. Ориентированным взвешенным графом назовем набор точек, некоторые пары которых соединены стрелками. По стрелке можно пройти из одной точки в другую. Будем далее называть точки вершинами, а стрелки — дугами. Каждая дуга имеет длину — некоторое положительное число. Длиной пути из вершины *A* в вершину *B* называется сумма длин всех дуг пути, ведущего из *A* в *B*. Задача о наименьшем пути состоит в нахождении пути из *A* в *B* с наименьшей длиной. Граф удобно хранить в виде матрицы смежности.



Составим следующий граф: вершинами графа назовем точки (0, 0), (*N*, *M*) и все точки вида (*K* – 1, *L* – 1) – (*K*, *L*), если квартал (*K*, *L*) можно срезать. Будем считать, что из точки (*K*1, *L*1) в точку (*K*2, *L*2) ведет дуга, если *K*1  *K*2 и *L*1 *L*2. Длина этой дуги равна (*K*2 + *L*2 – *K*1 – *L*1)·100. Кроме того, заменим длину дуг, соответствующих срезаемым кварталам, с 200 на . В таком графе нужно найти кратчайший путь из вершины (0, 0) в (*N*, *M*).

Формирование графа выглядит так:

**for** i := 1 **to** 2\*k + 2 **do**

**for** j := 1 **to** 2\*k + 2 **do**

    matr[i,j] := infinity;

**for** i := 1 **to** k **do**

  readln(cutx[i],cuty[i]);

**for**i := k + 1 **to** 2\*k **do**

**begin**

    cutx[i] := cutx[i - k] - 1;

    cuty[i] := cuty[i - k] - 1

**end**;

**for** i := 1 **to** 2\*k **do**

**for** j := 1 **to** 2\*k **do**

**if** (cutx[i] <= cutx[j]) **and**

        (cuty[i] <= cuty[j]) **then**

        matr[i,j] := (cutx[j] - cutx[i] +

        cuty[j] - cuty[i]) \* 100;

**for** i := 1 **to** k **do**

matr[i + k,i] := 141.42135;

cutx[2\*k + 1] := 0;

cuty[2\*k + 1] := 0;

cutx[2\*k + 2] := n;

cuty[2\*k + 2] := m;

**for** i := 1 **to** 2\*k + 1 **do**

   matr[2\*k + 1,i] := (cutx[i] + cuty[i])\*100;

**for** i := 1 **to** 2\*k + 2 **do**

    matr[i,2\*k + 2] := (m + n - cutx[i] - cuty[i])\*100;

Константу *infinity* положим равной 107. Теперь, когда матрица смежности заполнена, задача принимает вид стандартной задачи поиска кратчайшего пути в графе. Приведем здесь алгоритм Форда — Беллмана, решающий эту задачу. Этот алгоритм не является самым быстрым для подобной задачи, но, пожалуй, является самым простым. Идея алгоритма такова: будем работать в одномерном массиве *dist*[*i*], обозначающем длину текущего самого короткого пути из (0, 0) в*cutx*[*i*]. Будем пробегать все пары (*i*, *j*). В случае, если (длина текущего пути до*i*) + (длина дуги из *i* в *j*) меньше, чем длина текущего самого короткого пути до*j*, то изменим значение *dist*[*j*]. Этот процесс будем повторять, пока в массиве *dist*что-то будет меняться. Вот реализация этого алгоритма:

**for** i := 1 **to** 2\*k + 2 **do**

  dist[i] := infinity;

  dist[2\*k + 1] := 0.0;

  ind := true;

**while** ind **do**

**begin**

  ind := false;

**for** i := 1 **to** 2\*k + 2 **do**

**for** j := 1 **to** 2\*k + 2 **do**

**if** dist[i] + matr[i,j] < dist[j]

**begin**

             ind := true;

            dist[j] := dist[i] + matr[i,j]

**end**

**end**;

writeln(dist[2\*k + 2]:0:0);

***Задача “Круговая порука”***

**(VI командное соревнование школьников Свердловской области по программированию, 2002 г.)**

*Автор задачи — Л.Волков
Автор разбора — А.Ипатов*

Кто-то из *N* мальчиков и девочек опять разбил мамину любимую кружку. Мама рассердилась и перенумеровала мальчиков и девочек числами от 1 до *N*. После этого она подошла к мальчику (или девочке) под номером 1 и грозно спросила: “Кто разбил кружку?” Он (или она) честно ответил: “Я разбил кружку!” И был строго наказан.

Вы, конечно, понимаете, что рассказанная выше история идеализирована. На самом деле (правда это была или неправда — мы не знаем) мальчик (или девочка) под номером 1 ответил: “Да это не я! Это девочка (или мальчик) под номером *K*1!”. Потом мама подошла к номеру 2 и задала тот же вопрос…

Некоторые дети пытались ответить маме честно. Некоторые отвечали просто так, лишь бы что-нибудь сказать. Некоторые же заранее сговорились не выдавать маме преступника: каждый из группы юных заговорщиков называл кого-то другого — по кругу. В итоге мама совсем замучилась. Отчаявшись запомнить, что же говорил по поводу чашки каждый из отпрысков, она записала все их “показания” на листочек. Теперь она собирается серьезно расследовать дело. В первую очередь она решила установить, не сговорились ли какие-нибудь мальчики и девочки по кругу, так, как это было описано выше. Вы должны написать программу, чтобы помочь ей в такой ситуации, — ведь, судя по всему, эта разбитая чашка не первая и не последняя…

**Формат входных данных**

В первой строке входного файла записано число *T* (0 < *T* *<*1001) — количество тестов в файле. Каждый тест состоит из двух строк: в первой строке указано число *N* — количество детей (0 < *N* < 25 001). Во второй строке через пробел записаны *N*чисел — показания опрошенных детей. Мама записывала в эту строку на *i*-е место номер того ребенка, которого *i*-й ребенок назвал виноватым, или число 0, если *i*-й ребенок вдруг признавался в том, что он разбил кружку.

**Формат выходных данных**

В выходной файл следует вывести по одной строке для каждого теста. В эту строку надо записать YES, если показания детей по крайней мере выглядят непротиворечивыми: ровно один ребенок признался в том, что он разбил кружку, и нет группы детей, которая указывает друг на друга по кругу. В противном случае необходимо вывести NO.

**Пример**



**Решение**

В случае если массив показаний детей, назовем его *evd*, содержит больше одного нуля или не содержит их вовсе, то показания нужно считать противоречивыми. Рассмотрим основной случай, где ровно один ребенок признался в том, что разбил кружку. Каждый из остальных детей указал на кого-то другого. Рассмотрим ориентированный граф, в котором вершины обозначают детей, а дуги — их показания. Наша задача — проверить, содержит ли этот граф хотя бы один цикл.



Поскольку каждый ребенок указывает ровно на одного, данный граф можно заменить неориентированным, имея в виду, что один граф не содержит циклов тогда и только тогда, когда и другой не содержит их. В теории графов доказывается следующая теорема: граф из *n* – 1 ребра и *n* вершин не содержит циклов тогда и только тогда, когда является связным. Таким образом, если граф является связным, то ответ задачи — YES, иначе — NO. Алгоритм проверки графа на связность был описан в решении задачи “День объединения”.

Но мы сейчас рассмотрим другой алгоритм решения этой задачи, не требующий введения графа вовсе. Рассмотрим любого из детей, не сознавшихся в содеянном поступке. Покрасим его в первый цвет и перейдем к ребенку, на которого тот указывает. Покрасим его в такой же цвет и будем продолжать этот процесс, пока не дойдем до ребенка, который сознался, либо не встретим ребенка, уже покрашенного в первый цвет. Во втором случае в показаниях детей имеется цикл, и можно выдавать ответ NO. В первом случае найдем еще не покрашенного ребенка, покрасим его во второй цвет и продолжим аналогичный процесс окрашивания детей во второй цвет, пока не встретим ранее покрашенного ребенка (возможно, в первый цвет). В этом случае вновь найдем ребенка, не покрашенного ни в один из цветов, и начнем красить детей в третий цвет. Алгоритм закончит работу либо в случае, если все дети уже покрашены в какой-либо из цветов, либо в процессе окрашивания в цвет *i* мы дошли до ребенка, уже покрашенного в этот цвет.
В первом случае ответ задачи — YES, во втором — NO.

Цвет ребенка будем отмечать тоже в массиве *evd*, но отрицательными числами: –1, –2 и т.д. Приведем реализацию этого алгоритма.

read(N);

nulls := 0;

**for** j := 1 **to** N **do**

**begin**

    read(evd[j]);

**if** (evd[j] = 0) **then**

         inc(nulls)

**end**;

**if** nulls <> 1 **then**

  writeln('NO')

**else**

**begin**

   ind := true;

   color := -1;

**while** ind **do**

**begin**

         p := 1;

**while** (evd[p] <= 0) **and** (p <= N) **do**

               inc(p);

**if** p > N **then** **break**;

**while** evd[p] > 0 **do**

**begin**

                  p1 := evd[p];

                  evd[p] := color;

                  p := p1

**end**;

         ind := (evd[p] = color);

         dec(color)

**end**;

**if** ind **then** writeln('NO')

**else**writeln('YES')

**end**;

Для решения задачи достаточно вызвать алгоритм *T* раз.

***Задача “Монополия”***

**(Московская олимпиада по информатике 2005–2006 уч. г.)**

*Автор задачи и разбора — А.П. Лахно*

В Тридесятом государстве есть *N* фирм, занимающихся разработкой программного обеспечения. Однажды известный олигарх Тридесятого государства Иванушка решил монополизировать эту отрасль. Для этого он хочет купить максимальное число программистских фирм Тридесятого государства.

Он разослал предложения всем *N* компаниям и через некоторое время получил от каждой из них согласие или отказ. Однако он знает, что в бизнесе очень многое зависит от взаимного доверия партнеров.

В результате небольшого исследования Иванушка установил, между какими компаниями существует взаимное доверие (причем всегда если компания *A*доверяет компании *B*, то компания *B* доверяет компании *A*).

Теперь при желании Иванушка может повторно разослать предложения всем компаниям, включив в письма список компаний, давших согласие участвовать в его проекте. При этом каждая компания, независимо от своего первоначального мнения, дает согласие, если в списке есть хотя бы одна компания, которой она доверяет, и отказ в противном случае. Таким образом, некоторые компании, которые изначально не согласились участвовать в проекте, могут теперь дать свое согласие, а некоторые из давших согласие — наоборот, отказаться. В результате этого у Иванушки формируется новый список, который он опять может разослать фирмам. Он может сколь угодно долго повторять операцию, каждый раз рассылая текущий список. Иванушка может остановить процесс в любой момент и заключить договора с теми, кто после последней рассылки дал согласие.

Напишите программу, которая определит, какое максимальное число компаний может объединить Иванушка под своим началом.

Будем считать, что Иванушка — честный предприниматель и он никогда не подтасовывает рассылаемые им списки.

**Формат входных данных**

В первой строке входного файла записано число *N* — количество фирм (1  *N*  2000). Далее идут *N* чисел, описывающих ответ фирмы на первое предложение Иванушки (1 — согласие, 0 — отказ). Далее записано число *M* (0  *M*  200 000) — количество пар компаний, между которыми существует доверие. Далее записано *M* пар чисел, задающих номера фирм, между которыми существует взаимное доверие.

**Формат выходных данных**

Выведите в выходной файл одно число — максимальное число фирм, которое сможет купить Иванушка.

**Примеры**



**Решение**

Ситуация, рассматриваемая в задаче, описывается неориентированным графом, вершинами которого являются фирмы, а ребра задаются отношением взаимного доверия.

Изначально вершины, соответствующие согласившимся фирмам, отмечены. Далее, на каждой итерации создается новый список отмеченных вершин: в него включаются те и только те вершины, у которых на предыдущем шаге была хотя бы одна отмеченная смежная вершина.

Рассмотрим ключевые закономерности изменения списка отмеченных вершин.

Пусть на некотором шаге имеются две смежные вершины, хотя бы одна из которых отмечена. Тогда после одной итерации отмеченная вершина может “погаснуть”, но после второй она обязательно вновь станет отмеченной. Заметим, что эта вершина заведомо будет попадать в список отмеченных через каждые две итерации.

Если же имеются две отмеченные смежные вершины, то они останутся отмеченными после любого числа итераций.

**Первое решение**

Основываясь на этих закономерностях, приходим к следующему алгоритму решения задачи. Раскрасим имеющийся граф в два цвета с помощью поиска в глубину: первую вершину каждой компоненты связности красим в белый, все смежные с ней — в черный, все смежные с ними — снова в белый и т.д. Если при этом в какой-то компоненте связности есть цикл нечетной длины, то возникает коллизия: компоненту связности нельзя раскрасить в два цвета так, чтобы все вершины, смежные с белыми, были черными, а все вершины, смежные с черными, — белыми.

Параллельно с раскраской для каждой вершины запоминаем номер компоненты связности, в которой она лежит. Для компонент связности, содержащих циклы нечетной длины, помечаем, что их раскрасить не удалось. Кроме того, для каждой компоненты связности запоминаем число белых вершин и число черных вершин в ней.

Проходим по первоначальному списку отмеченных вершин.

Если компонента связности, в которой лежит отмеченная вершина, содержит циклы нечетной длины, то, начиная с некоторой итерации, все вершины компоненты будут попадать в список отмеченных на каждом шаге.

Если исходный список отмеченных вершин содержит как белую, так и черную вершину какой-то компоненты связности, то, так же, как и в предыдущем случае, начиная с некоторой итерации, все вершины компоненты будут попадать в список отмеченных на каждом шаге.

Если список вообще не содержит отмеченных вершин из какой-то компоненты, то ни одна вершина этой компоненты никогда не попадет в список отмеченных.

Для остальных компонент в первоначальном списке содержатся отмеченные вершины только одного цвета (для каждой компоненты — своего). Начиная с некоторой итерации, на каждом четном шаге в список отмеченных попадают вершины таких компонент, отмеченных тем же цветом, что и отмеченные вершины в исходном списке, а на нечетных шагах — вершины противоположного цвета. Единственным исключением здесь являются компоненты связности, состоящие ровно из одной вершины, — эта вершина в любом случае никогда не попадет в список отмеченных после первого шага.

С учетом всего вышесказанного, ответ задачи вычисляется как максимум из трех чисел: количества отмеченных вершин в исходном списке и количеств отмеченных вершин на четных и нечетных шагах с достаточно большим номером, которые, в свою очередь, несложно вычисляются с помощью ранее посчитанных величин, указанных в начале разбора.

**Второе решение**

Приведем вариант решения, несколько менее естественного, но существенно более изящного. Построим граф, вершинами которого являются пары вида (четность итерации, номер фирмы), а ребра соединяют вершины с разной четностью, соответствующие взаимно доверяющим фирмам. Пометим исходно вершины, соответствующие четной итерации и изначально согласившимся фирмам. Запустим теперь из помеченных вершин обход в глубину или ширину. Ответом задачи будет максимум из количества изначально согласившихся фирм и количеств обойденных вершин с четным и нечетным числом итераций соответственно.

Асимптотика обоих решений определяется асимптотикой процедуры обхода графа, т.е. *O*(*N* + *M*).

В ограничениях задачи было достаточно разумно при отсутствии правильного решения написать по крайней мере решение, моделирующее процесс. При использовании отсечения по времени и аккуратном кодировании такое решение тоже набирало полный балл.

Заметим, что при моделировании, как это ни странно, не достаточно *N*итераций. Приведем простой пример входных данных: цепочка из пяти вершин, у которой, кроме того, соединены пятая и третья вершины. Изначально отмечена только первая вершина:



В данном случае ответ достигается лишь на шестой итерации при пяти вершинах. В общем же случае оптимальное решение будет найдено не более чем за 2*N* итераций.

***Задача “Роботы”***

**(Московская командная олимпиада по информатике 2005–2006 уч. г.)**

*Автор задачи — Е.В. Андреева
Автор разбора — А.В. Фонарев*

В подземелье есть *N* залов, соединенных туннелями.
В некоторых залах находятся роботы, которые одновременно получили команду собраться в одном месте.

Роботы устроены так, что, получив команду, они все начали двигаться с такой скоростью, что туннель между двумя любыми залами любой робот преодолевает за 1 минуту. Роботы не могут останавливаться (в том числе и в залах), а также менять направление движения, находясь в туннелях. Однако, попав в зал, робот может из него пойти по тому же туннелю, по которому он пришел в этот зал.

Напишите программу, вычисляющую, через какое минимальное время все роботы смогут собраться вместе (в зале или в туннеле).

**Формат входных данных**

Во входном файле записаны сначала числа *N* — количество залов (1  *N*Ј 400) и *K* — количество туннелей. Далее записано *K* пар чисел, каждая пара описывает номера залов, соединяемых туннелем (по туннелю можно перемещаться в обе стороны). Между двумя залами возможно несколько туннелей. Туннель может соединять зал с самим собой. Далее записано число *M*(1  *M* 20 000) — количество роботов. Затем идет *M* чисел, задающих номера залов, где вначале расположены роботы. В одном зале может быть несколько роботов.

**Формат выходных данных**

В выходной файл выведите минимальное время в минутах, через которое роботы могут собраться вместе. Если роботы никогда не смогут собраться вместе, выведите одно число –1 (минус один).

**Примеры**





**Решение**

Переведем задачу на язык графов. Вершинами нашего графа будут залы, а ребрами — туннели. Граф взвешенный, вес каждого ребра — 1. Тогда роботам требуется за кратчайшее время встретиться в вершине или на ребре.

Сразу отметим два факта:

1. Если робот может прийти в вершину за время *t*, он может повторно попасть туда же за время *t* *+* 2.

2. Если роботы встречаются в тоннеле (на ребре), то это происходит в середине, причем часть роботов идут по ребру в одну сторону, а часть в другую (иначе они бы уже встретились в одной вершине).

Теперь становится понятно, как найти минимальное время, за которое роботы могут собраться в данной вершине *v*: смотрим, за какое наименьшее четное (нечетное) время роботы придут в *v* (это наибольшая из длин кратчайших четных (нечетных) путей каждого робота до *v*). Первое, что приходит в голову, — разбить каждое ребро на две равные части, т.е. добавить вершины, которые будут соответствовать серединкам. Но мы так делать не будем, т.к. это решение требует некоторой аккуратности. Если мы знаем время *Ti*, за которое данный робот добирается до вершины *i*, то до центра ребра *uv*робот доберется за *min*(*Tu*, *Tv*) *+* 1/2. Появляется следующая идея: для каждого робота найдем кратчайшие четный и нечетный пути до всех вершин графа. После чего для каждой вершины найдем время сбора в ней. То же самое сделаем с каждым ребром (сбор в центре) и возьмем общий минимум. Если собраться нигде не удалось (ясно, что это бывает только в случае, когда найдутся два робота в разных компонентах связности графа), ответом будет –1.

Самый интересный момент решения — поиск кратчайших путей четной и нечетной длины. Для этого немножко модифицируем обход в ширину, вычисляя сразу 2 величины — четное и нечетное расстояния. Граф будем хранить в виде списков смежных вершин. При считывании входных данных можно сразу удалить кратные ребра, но ни в коем случае нельзя удалять петли (если в вершине есть петля, это позволяет роботу вернуться туда же не за 2 хода, а за 1). Все расстояния в задаче полуцелые (т.е. становятся целыми при умножении на 2), а значит, если считать, что вес ребра — 2, то все вычисления можно проводить с помощью целых типов данных (это дает хороший выигрыш во времени). В конце же просто разделим ответ на 2.

Теперь можно привести основной фрагмент решения задачи.

**var** m: **array**[1..400,1..400, 0..1] **of** integer;

l: **array**[1..400,0..400] **of** integer;

n, k, nr, i, j, t: integer;

r: **array**[1..400] **of** integer;

a, b: **array**[1..20000] **of** integer;

*m*[*i*][*j*][*k*] — матрица кратчайших путей: *i* — вершина, из которой мы ищем путь, *j*— пункт назначения, *k* — четность пути (0 или 1); *l*[*i*][*j*] — список смежных вершин: *i* — вершина, для которой хранится список, *l*[*i*] — сам список (*l*[*i*][*0*] — его длина); *a*[*i*] и *b*[*i*] — концы ребер.

Далее описан модифицированный обход графа в ширину. Он использует очередь, реализованную при помощи массива *o*. Если путь данной четности до вершины не существует, значение соответствующей ячейки будет равно бесконечности (число, большее 106). За одну итерацию цикла из очереди берется очередная вершина, обновляются расстояния для смежных вершин. Те, для которых расстояния обновились и которые не находятся в очереди, туда заносятся. Массив *u* показывает, находится ли данная вершина в очереди.

**procedure** bfs(v: integer);

**var** p1, p2, cv: integer;

**begin**

  fillchar(u, sizeof(u), 0);

  p1 := 1;

  p2 := 1;

  o[p1] := v;

  u[v] := true;

**while** p1 <= p2 **do begin**

    cv := o[p1];

    inc(p1);

    u[cv] := false;

**for** i := 1 **to** l[cv][0] **do begin**

**if** m[v][l[cv][i]][0] > m[v][cv][1] + 2 **then**

**begin**

            m[v][l[cv][i]][0] := m[v][cv][1] + 2;

**if not** u[l[cv][i]] **then begin**

                u[l[cv][i]] := true;

                inc(p2);

                o[p2] := l[cv][i]

**end**

**end**;

**if** m[v][l[cv][i]][1] > m[v][cv][0] + 2 **then**

**begin**

             m[v][l[cv][i]][1] := m[v][cv][0] + 2;

             **if** not u[l[cv][i]] then **begin**

               u[l[cv][i]] := true;

               inc(p2);

               o[p2] := l[cv][i]

**end**

**end**

**end**

  end

**end;**

***Вычислительная геометрия***

***Задача “Найди прямую”***

**(Московская олимпиада по информатике 2005–2006 уч. г.)**

*Автор задачи и разбора — Б.О. Василевский*

На плоскости дано множество отрезков. Требуется найти прямую, которая пересекла бы наибольшее возможное количество из данных отрезков и при этом проходила бы как минимум через две точки с целочисленными координатами.

Считается, что прямая пересекает отрезок, если она имеет с ним хотя бы одну общую точку (т.е. она может проходить через конец отрезка, внутреннюю точку отрезка, либо содержать весь отрезок).

**Формат входных данных**

Во входном файле записано сначала число *N* — количество отрезков (1  *N* 1000). Далее идет *N* четверок чисел *Xi*1, *Yi*1, *Xi*2, *Yi*2, задающих координаты концов отрезков. Все эти числа целые, по модулю не превосходящие 10 000.

Заданные отрезки могут пересекаться, иметь общие части, один из них может полностью содержаться внутри другого. Отрезки имеют ненулевую длину.

**Формат выходных данных**

В выходной файл выведите координаты каких-нибудь двух точек, через которые проходит прямая, пересекающая наибольшее количество отрезков. Координаты точек должны быть целыми и не должны по модулю превышать 107.

**Примеры**



**Решение**

*Степенью*прямой назовем количество данных отрезков, которые она пересекает; концы отрезков будем называть *вершинами*.

Прежде всего обсудим, каким может быть относительное расположение искомой прямой и данных отрезков. Рассмотрим прямую с максимальной степенью.

Если прямая не горизонтальная, будем двигать ее влево горизонтально, пока она не пройдет через какую-нибудь вершину. Иначе можно двигать ее вертикально вверх. Новая прямая будет иметь такую же степень, как и начальная прямая, так как количество пересечений не может увеличиться, а уменьшается оно только при прохождении через концы отрезков.

Итак, можно считать, что искомая прямая должна проходить через какую-либо вершину. Теперь будем вращать эту прямую против часовой стрелки вокруг вершины, через которую она проходит, пока прямая не пройдет через вторую вершину. Аналогично, степень полученной прямой равна степени исходной. И теперь прямая проходит через две различных вершины.

На этих идеях основано решение с алгоритмической сложностью *O*(*N*3), где *N* — количество отрезков: перебираем все пары вершин; для каждой пары считаем степень прямой, через эти вершины проходящей; среди них выбираем прямую с наибольшей степенью. Но такое решение не набирает полного балла.

Решение с алгоритмической сложностью *O*(*N*2log*N*) использует только первое рассуждение: искомая прямая проходит через вершину. Таким образом, если для каждой вершины найти прямую наибольшей степени, которая проходит через нее, то, выбрав среди всех степеней максимум, получим ответ.

Научимся искать прямую с наибольшей степенью, проходящую через вершину*P*(*xp*, *yp*). Любое упоминание прямой теперь подразумевает, что она проходит через *P*. Сразу договоримся не рассматривать отрезки, проходящие через *P*, так как они не влияют на поиск прямой максимальной степени. Главное — не забыть их подсчитать и учесть при вычислении степени прямой.

Общая идея такова: будем вращать прямую *y* = *yp* вокруг *P* против часовой стрелки, подсчитывая пересечения.

*Ориентированной площадью*[A*,*B] пары векторов A = (*x*1, *y*1) и B = (*x*2, *y*2) назовем число, равное *x*1· *y*2 – *y*1·*x*2. По абсолютной величине она равна площади параллелограмма, у которого в качестве сторон взяты A и B, отложенные от одной точки. Она равна нулю тогда и только тогда, когда A и B коллинеарны. В противном случае знак [A, B] дает нам информацию о направлении кратчайшего поворота от вектора A к вектору B. Если система координат выбрана стандартным образом, то [A, B] больше нуля тогда и только тогда, когда кратчайший поворот — против часовой стрелки, см. *рис.* 1 (соответственно, [A, B] меньше нуля тогда и только тогда, когда кратчайший поворот — по часовой стрелке, см. *рис.* 2). В литературе по вычислительной геометрии эту же величину часто называют *векторным произведением* векторов A и B.



*Рис.* 1 *Рис.* 2

Назовем *началом* отрезка *XY* ту его вершину, которая встретится при вращении нашей прямой раньше остальных точек этого отрезка; *концом* — ту, которая встретится последней. Это можно определить, вычислив знак ориентированной площади *S* = [*PX*, *PY*]: если *X* — начало, то *S* > 0. В случае, если *X*, *Y* и *P* лежат на одной прямой, началом будем считать любую из вершин, концом — оставшуюся.

Рассмотрим для начала более простой случай: по отношению к *P* все отрезки лежат справа и сверху. Чтобы реализовать “вращение прямой”, надо отсортировать все вершины по возрастанию *полярного угла* (то есть по углу между лучом из точки *P* в вершину и осью абсцисс), в случае совпадения будем считать, что конец всегда больше начала.

Cделаем небольшие пояснения. Сортировка точек по полярному углу проходит почти так же, как и сортировка чисел, только для сравнения двух точек *X*, *Y*используется знак ориентированной площади *S* = [*PX*, *PY*].
“*X* меньше *Y*”, если *S* > 0 (“*X* больше *Y*”, если *S* < 0).

Прямая *y* = *yp* не пересекает ни одного отрезка в силу того, что все концы отрезков лежат выше нее.

*Шаг 1.* Считаем переменную *C* равной 0.

*Шаг 2.* Просматриваем все вершины так, как они идут в отсортированном списке.

*Для каждой вершины Q (*

*Если Q — начало, то (*

*Увеличиваем C на 1.*

*Если C больше найденного максимума,*

*то обновляем максимум, запоминаем,*

*что соответствующая прямая проходит*

*через Q.*

*) иначе уменьшаем C на 1.*

*)*

*)*

Понятно, что после просмотра вершины *Q* степень всех прямых между *PQ* и *PS*(где *S* — следующая за *Q* вершина в отсортированном списке) равна *C* (любая из этих прямых пересекает все отрезки, которые уже “начались”, но еще не “закончились”, их ровно *C* по построению). Среди всех таких значений *C* надо выбрать наибольшее. В качестве прямой с соответствующей степенью можно брать, к примеру, *PQ*, как и описано в алгоритме.

Сложность общего случая (когда отрезки расположены произвольно) заключается в следующем: пусть мы сравниваем точки как обычно — по полярному углу. Тогда привычное свойство транзитивности сравнения “из *a* *b*, *b*  *c* следует *a*  *c*” не выполняется, например, для вершин *a* = (0, 1), *b* = (1/2, –1/2), *c* = (–1/2, –1/2), *P* = (0, 0). Поэтому любая сортировка за время*N*log*N*, основанная на таком свойстве, будет сортировать точки неправильно. А ведь именно за счет сортировки за время *NlogN* мы имеем оценку *O*(*N*2log*N*), а не*O*(*N*3).

Существует несколько способов выйти из этой ситуации, но наиболее простой, на наш взгляд, следующий.

Исходная прямая — по-прежнему *y* = *yp*. Разделим отрезки на два типа:

1) пересекающие *y* = *yp*,

2) не пересекающие *y* = *yp*.

Идея в том, чтобы перед сортировкой подвергать симметрии относительно *P* все вершины, лежащие ниже *P* (не забывая, начало это или конец). Но делать такую симметрию надо аккуратно.

Пусть отрезок *XY* после возможной симметрии его вершин, описанной выше, перешел в отрезок *X*'*Y*'. Если ориентированная площадь [*PX*, *PY*] отличается по знаку от ориентированной площади [*PX*', *PY*'], то визуально начало и конец “поменялись местами” (мы по-прежнему называем *X*' началом, если *X* — начало, и, наоборот, — концом, если *X* — конец). То есть при вращении наша прямая сначала встретит конец *X*'*Y*', а потом начало (см. *рис.* 3). В этом случае надо учитывать, что прямые из области 3 (см. *рис*. 3) тоже пересекают *XY* (в алгоритме — шаг 3). Кстати, такое возможно, только если *XY* — первого типа.



*Рис.* 3

С отрезками второго типа таких неудобств не будет.

*Замечание 1.* Если приступать к сортировке, проделав предварительно лишь описанные преобразования, мы не получим правильного порядка. Пусть, например, *P*(0, 0), *A*(–1, 0), *B*(1, 0), *A* и *B* — начала. Тогда вершины *A* и *B* будут считаться равными (подумайте, почему). Чтобы избежать этого, можно подвергать симметрии еще и точки, лежащие на *y* = *yp*, но левее *P*. Тогда никакие две вершины не будут лежать с *P* на одной прямой и одновременно по разные стороны относительно *P*.

*Замечание 2.* Чтобы проверить, лежит ли *P* на отрезке *AB*, надо сначала убедиться, что все эти три точки лежат на одной прямой. Если это верно, можно посмотреть скалярное произведение векторов *PA* и *PB*: если оно меньше или равно 0, то *P* принадлежит *AB* (0 может быть, если один из векторов нулевой), в противном случае — нет.

Для каждой точки *X* через *X*' будем обозначать следующее: если *X* ниже *P* или *X*принадлежит *y* = *yp* и левее *P*, то *X*' симметрична *X* относительно *P*. В противном случае *X* = *X*'.

Чтобы алгоритм получился полным, вспомним о существовании отрезков, проходящих через *P*. По-прежнему считаем, что концы отрезка ему принадлежат.

*Шаг 1.* Считаем переменную *C* равной 0. Список для сортировки считаем пустым.

*Шаг 2 (создание списка для сортировки).*

*Для каждого отрезка XY (*

*Если P принадлежит XY, то (*

*Увеличить C на 1*

*) иначе (*

*Если XY — первого типа, то (*

*Если знаки [PX, PY] и [PX', PY']*

*различаются (оба числа должны быть отличны от нуля),*

*то увеличить C на 1.*

*)*

*)*

*Пусть S = [PX, PY].*

*Если S >= 0, то X' помечаем как начало, Y' — как конец.*

*Если S < 0, то Y' помечаем как начало, X' — как конец.*

*Добавляем в список X'*

*Добавляем в список Y'*

*)*

*)*

*Шаг 3 (сортировка и поиск прямой с максимальной степенью).* Теперь отсортируем полученный список по полярному углу и далее будем действовать, как в простом случае (единственное отличие — стартовое значение *C* может быть отлично от нуля).

***Задача “Разрезание многоугольника”***

**(I городская олимпиада, г. Нижний Новгород, 2004–2005 уч. г.)**

На плоскости заданы многоугольник и прямая; прямая не проходит через вершины многоугольника.

Напишите программу, которая определит, на сколько частей делит эта прямая многоугольник.

Формат входных данных

В первой строке входного файла находится одно число *N* (3  *N* 1025) — число вершин многоугольника. Далее следуют *N* строк, задающие вершины многоугольника: в *i*-й из этих строк находятся два целых числа *xi* и *yi* (|*xi*|, |*yi*|  10 000) — координаты *i*-й вершины многоугольника. Вершины занумерованы от 1 до *N* в порядке обхода против часовой стрелки.

В (*N* + 2)-й строке входного файла находятся четыре целых числа *px*, *py*, *qx*, *qy*, задающие прямую (все числа не превосходят по модулю 10 000). Прямая проходит через точки (*px*, *py*) и (*qx*, *qy*); эти точки различны.

**Формат выходных данных**

В выходной файл выведите одно целое число — ответ на задачу.

**Пример**

|  |  |
| --- | --- |
| **Входной файл** | **Выходной файл** |
| 4 | 2 |
| 0 0 |   |
| 1 0 |   |
| 1 1 |   |
| 0 1 |   |
| -1 -1 3 4 |   |

**Решение**

В этой задаче требовалось найти количество частей, на которые прямая разделяет многоугольник.

Очевидным образом количество точек пересечения прямой и границы многоугольника четно. Действительно, ни одна вершина, а следовательно, ни одна сторона многоугольника не лежит на секущей прямой. Кроме того, если мы будем двигаться по прямой из бесконечности, то сколько раз мы “войдем” в многоугольник, то есть пересечем его границу, проходя извне внутрь, столько же раз и “выйдем”, то есть пересечем его границу, проходя изнутри вовне.

Ясно, что каждая пара “вход–выход” в/из многоугольника увеличивает количество частей на 1. Так как сначала часть была одна — сам многоугольник, то окончательная формула для количества частей следующая: (*количество пересечений*)/2 + 1.

Теперь посчитаем количество пересечений. Проще всего для каждого ребра многоугольника проверить, лежат ли его концы по разные стороны от прямой. Таким образом, для каждого ребра (*xi*; *yi*)–(*xi+1*; *yi+1*) и прямой, заданной двумя точками (*px*; *py*) и (*qx*; *qy*), имеем условие того, что точки *Ai* (*xi*; *yi*) и *Ai+1*(*xi+1*;*yi+1*) лежат по разные стороны от прямой *P*(*px*; *py*)–*Q*(*qx*; *qy*): векторные произведения векторов *PAi*(*xi* – *px*; *yi* – *py*) и *PAi+1*(*xi+1* – *px*; *yi+1* – *py*) на вектор*PQ*(*qx* – *px*; *qy* – *py*) имеют разные знаки:

[*PAi*, *PQ*] x [*PAi+1*, *PQ*] < 0.

Но в программировании именно такой способ проверять, что величины имеют разный знак, очень часто является ошибкой. Дело в том, что каждое векторное произведение в нашей задаче по модулю может достигать 800 000 000, то есть их произведение уже не влезет в четырехбайтовый целый тип данных. В данном случае лучше записать более длинное условие: либо первое произведение меньше нуля, а второе больше, либо наоборот. Другой способ состоит в том, чтобы вместо векторных произведений перемножать их знаки.

***Задача “Точность попадания снаряда”***

**(VI командное соревнование школьников Свердловской области по программированию, 2002 г.)**

Автор задачи — А.Ботов
Автор разбора — А.Ипатов

“*Знаете ли вы, что неточность попадания снаряда можно компенсировать его диаметром?”*

*С.В. Сизый*

В этой задаче вам предлагается с помощью компьютера определить самый маленький диаметр, которым можно компенсировать неточность попадания снаряда в каждом конкретном случае. Будем считать, что все цели являются выпуклыми многоугольниками. Попаданием считается ситуация, когда круглая воронка, остающаяся от снаряда, задевает хотя бы одну точку цели (диаметр воронки равен диаметру снаряда).



**Формат входных данных**

В первой строке входного файла находятся 3 числа — координаты попадания центра снаряда и количество сторон многоугольника *N*(3  *N* 100), следующие *N* строк содержат координаты его вершин, перечисленные по часовой стрелке. Все координаты являются целыми числами из диапазона [–1 000 000; 1 000 000].

**Формат выходных данных**

В выходном файле должно быть записано единственное число — минимальный диаметр снаряда, который поразит цель. Число следует округлить до трех знаков после запятой.

**Пример**



**Решение**

Первое, что нам нужно научиться определять, — это лежит ли центр снаряда в мишени. Если это так, то ответ задачи — 0.000. Если нет, то расстояние от точки до многоугольника равно минимуму расстояний от этой точки до его сторон.



Обозначим центр снаряда — *O*, мишень — *A*1*A*2…*AN* и рассмотрим ориентированные углы *A*1*OA*2, *A*2*OA*3, …, *A*N-1*OA*N, *A*N*OA*1. Они все имеют одинаковые знаки (поворот между лучами происходит везде по часовой стрелке либо везде — против нее) тогда и только тогда, когда точка лежит внутри многоугольника. На рисунке справа все углы ориентированы против часовой стрелки, а на рисунке слева — угол *A*1*OA*2 ориентирован по часовой стрелке, а угол *A*5*OA*1 — против. Ориентацию угла легко определить с помощью векторного произведения векторов *OA*i и *OA*i+1.

Приведем код функции, определяющей, лежит ли точка внутри многоугольника.

**function** inside: boolean;

**var** i: integer; ind: boolean;

**begin**

  ind := true;

**for** i := 1 **to** n **do**

**if** ((x[i] - xx)\*(y[i + 1] - yy) -

       (x[i + 1] - xx)\*(y[i] - yy))\*

       ((x[i + 1] - xx)\*(y[i + 2] - yy) –

       (x[i + 2] - xx)\*(y[i + 1] - yy)) < 0

**then**ind := false;

  inside := ind

**end**

Здесь *x*[*i*], *y*[*i*] — координаты вершин многоугольника (*x*[*n*+ 1] = *x*[1], *x*[*n*+ 2] =*x*[2], *y*[*n*+ 1]=
= *y*[1], *y*[*n*+ 2] = *y*[2]), *xx*, *yy* — координаты точки.



Итак, пусть мы определили, что точка лежит вне многоугольника, и хотим вычислить расстояния от нее до сторон многоугольника. Рассмотрим произвольный отрезок *AB* и точку *C*, заданные координатами. Опустим перпендикуляр из *C* на прямую *AB*.



Если он попадает внутрь отрезка (ситуация на рисунке слева), то он и является кратчайшим расстоянием от *C* до *AB*. Иначе (см. рисунок cправа) расстояние до отрезка равно минимуму из длин *AC* и *BC*. Если оба угла *CAB* и *CBA* лежат в пределах [0; /2], то налицо первый случай, иначе — второй. Является ли угол тупым, легко узнать по знаку его косинуса, а тот может быть определен через скалярное произведение. Зная косинусы углов *CAB* и *CBA*, а также расстояния*AC* = *d*1 и *BC* = *d*2, вычислим расстояние от *C* до прямой *AB*.

Приведем код основной части программы.

mind := infinity;

**if** inside **then**mind := 0

else

**for** i := 1 **to** n **do**

**begin**

          a := (x[i] - x[i + 1])\*(xx - x[i + 1]) + (y[i] - y[i + 1])\*(yy - y[i + 1]);

          b := (x[i + 1] - x[i])\*(xx - x[i]) +(y[i + 1] - y[i])\*(yy - y[i]);

          d1 := sqrt(sqr(xx - x[i + 1]) +sqr(yy - y[i + 1]));

          d2 := sqrt(sqr(xx - x[i]) +sqr(yy - y[i]));

**if** a < 0 **then**d := d1

**else**

**if** b < 0 **then**d := d2

**else**

**begin**

               dt := a/sqrt(sqr(x[i + 1] - x[i]) +sqr(y[i + 1] - y[i]));

               d := sqrt(d1\*d1 - dt\*dt)

**end**;

**if** d < mind **then**mind := d

**end**;

writeln(2\*mind:0:3);

Здесь *a* и *b* — скалярные произведения (*AC*, *AB*) и (*BC*, *BA*) соответственно, *dt* — проекция вектора *AC* на *AB* (с помощью нее по теореме Пифагора найдем  *d* — расстояние от *C* до *AB*). *x*[*n*+ 1] = *x*[1], *y*[*n*+ 1] = *y*[1]. Константа *infinity*равна 108.